FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

```
//Příloha č. 2 Smlouvy - Technická specifikace softwarového rozhraní v ANSI C++
//
// Source code written by Czech Technical University to specify the conditions
// on the software interroperability for tender procedure. The goods is
// “6D collaborative robot”. Date: May 2021.
// In Czech (“Dodavka kolaborativniho 6D robot”)
//
// The programming language is ANSI C/C++. The operating system can be
// either OS Linux or both OS Linux and OS MS Windows. The example API should
// be fully available and functional upon the product delivery including the source
// code for API in ANSI C/C++. The API implementation cannot use any commercial
// programming software requiring additional license fees of any kind
// to the debit of Czech Technical University or other chargers and obligations.

#include <cstdlib>
#include <iostream>
#include <iostream>
#include <chrono>
#include <ctime>
#include <cassert>

// Include other libraries needed by the selling party to fullfill the
// condition of the tender procedure

// Use standard namespace
using namespace std;

// -----------------------------------------------------------------
// COBOT API (Application Programming Interface in ANSI C++)

// -----------------------------------------------------------------
// BEGIN OF API SPECIFICATION

// Connects to the cobot controller on a given IP address
// Returns 0 on success
// Returns -1 on failure
int
ConnectCobotCommunication(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  return 0; // success
}

// Disconnects the cobot controller on a given IP address
void
DisconnectCobotCommunication(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
}

// Set the parameters to optimize the motion: payload in kg
// and moment of intertia of payload in kg.m2 and the distance
// between the head of robot to the center of gravity of load.
int
SetWeightMomentOfInertia(char ipaddress[], float payloadWeight,
                   float kgm2, float distance)
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  if ((payloadWeight >= 0.0) &&
      (payloadWeight < 15.5) &&
      (kgm2 >= 0.0) &&
      (kgm2 <= 0.8) ) { // conditions if parameters are set correctly
    // Pass to the controller the three values:
    // a) payload weight,
    // b) payload center of gravity distance,
    // c) payload moment of inertia
    // .....

    return 0; // OK
  }
  return -1; // error - out of range
}

// Start movement of the cobotic arm the a new pose that was set by
// the command It is non-blocking operation, so it returns immediately
// even if the motion is in process
// Returns 0 if the operation was successfully started
// Returns -1 if the operation cannot be executed for some reason -
// - setup was not made yet Now start the movement of a cobot arm to a
// new pose, non-blocking operation
int
ExecuteCobotMotion(char ipaddress[], float eps)
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  // nonblocking operation execution

  // Execute the motion of a cobot to a new position of joints J06
  // with the maximum error at joints eps. The values are given in
  // degrees

  // returns 0 ... success, returns 1 ... failure, wrong joint
  // position required
  return 0;
}

// Stops running motion immediately, not resulting in an error state
// It is non-blocking operation execution
// Returns 0 on success
// Returns a negative value on error, the motion cannot be executed
int
StopCobotMotion(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  return 0; // returns 0 ... success
}

// Returns current cobot position at six joints at this moment of time
// Returns 0, if cobot is not moving now
// Returns 1, if cobot is moving now
// Returns -1 or other negative value if the cobot motion was not
// successfully completed,
// for example, there was a blocking obstacle on the motion path
int
GetCobotPosition(char ipaddress[], float J06[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  // Get J06[] just now, with the uncertainity of latency by UDP
  // communication if the cobot is moving or not

  // ....
  if (1)  // to be changed by correct condition
    return 0; // return 0 if cobot is not moving at this moment

  return 1; // return 1 if the cobot is moving now

}

// If a cobot arm/controller went to a failure state, it removes
// the error state and reinitate the status
int
ReinitiateCobotController(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  // Returns -1 or negative code if the operation failed

  return 0; // return 0 if the operation was ended with success
}

// Returns 0, if a cobot arm/controller has no error and is communicating
// Returns a negative value, when the cobotic arm or the controller
// went to a failure
int
IsCobotInErrorState(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // returns a value corresponding to some error state indicating
  // which problem has occurred.

  // For example, it can return -1, if the cobot motion was stopped
  // during motion by an emergency stop
  // It can return -2 if the cobot motion path was blocked by an
  // obstacle.
  return 0; // not in error state
}

// It is non-blocking operation execution, returns the status.
// Returns 0 if the cobot is not moving at this moment
// Returns 1 if the cobot is moving at this moment
// Returns -1 or other negative value if the last cobot motion
// operation was not successfully completed, for example, there was a
// blocking obstacle on the motion path
int
IsMoving(char ipaddress[])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  if (1)  // to be changed by correct condition
    return 0; // return 0 if cobot is not moving at this moment

  return 1; // return 1 if the cobot is moving at this moment

  // returns negative value if the cobot got to the error state
  if (0)
    return -1; // error state
```

```
}

// Set speed of motion at TCP (tool center point at the mid of flange)
// in degrees/sec
// Returns 0 ... on success, and negative value on failure, possibly
// indicating the problem
// It can be executed only when the cobot is not moving only.
int
SetSpeedJoints(char ipaddress[], float speed)
{
  if (speed < 0) return -1; // error

  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // ....
  return 0; // success
}

// Sets the speed of motion at TCP in m/s and and accel, decel in m/s^2.
// It has to be set before the motion and not during the motion.
// Returns 0 on success
// Returns -1 on failure, cobot is moving
// Returns -2 on failure, required speed is out of range
// Returns -3 on failure, required acceleration is out of range
// Returns -4 on failure, required deceleration is out of range
int
SetSpeedTCP(char ipaddress[], float speed, float accel, float decel)
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // ....
  return 0; // success
}

// This function converts the joints positions to TCP (tool center point at
// the midddle of the cobot head flange
// Returns 0 on success
// Returns -1 on faiulre ... unable to convert, joint position out of range
// J06[] specifies the joints of cobots - input
// TCP specifies the tool center point - position + euler angles,
// position is 3 values, rotation 3 values
int
ConvertJointsToTCP(char ipaddress[], float J06[6], float TCP[6])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // ....
  return 0; // success
}

// Converts the TCP (tool center point at midddle of the cobot head flange)
// to six joints position
// Returns 0 on success.
// Returns -1 on failure ... unable to convert, TCP in input is out of range
// TCP specifies the tool center point - position + euler angles (input)
// J06[] specifies the angles of joints of cobot in range <-180, 180> degrees
int
ConvertTCPtoJoints(char ipaddress[], float TCP[6], float J06[6])
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // ....
  return 0; // success
}

// Set the planned positions at joints for the next motion for the exact
// specification of time at that motion at this command
// Intermediate positions can be set if N>1, it then requires J[]
// array is of length N*6 The values in array timeC[] must be in range
// <0,1> and in ascending order.
//
// Returns 0 ... on success
// Returns -1 ... on failure, the position J[] was incorrectly specified
// Returns -2 ... on failure, the IP address was incorrectly specified
// Returns -3 ... wrong setting of timeC[] array, it is not an ascending order
// It can be sucessfully executed only if the last motion execution is finished.
int
SetPositionForTheNextMotion(
  char ipaddress[],
  int N, // how many positions, at least 1
  float timeC[], // event times at these positions
  float J[], // joints setting for the specified times at multiple of 6 values
  float &expectedTime)
{
  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  // ..........

  // Set the expected time for the whole planned motion for current setting
  expectedTime = 0; // TO BE IMPLEMENTED BY CONTRACTOR, this line is not correct

  return 0; // succcess
}


// Returns the planned positions at joints for the next motion for the exact
// specification of time at that motion at this command
// TimeC = 0.0  ... corresponds to the sitation when cobot was motion is started
// TimeC = 1.0  ... corresponds to the sitation when cobot was motion is finished
// Returns 0 ... on success
// Returns -1 ... on failure, the time was incorrectly specified
// Returns -2 ... on failure, the IP address was incorrectly specified
// It can be sucessfully executed only if the last motion execution is finished.
int
GetPlannedPositionForNextMotion(char ipaddress[], float timeC, float J[])
{
  if ((timeC<0)||(timeC>1.0))
    return -1; // the time event is out of range

  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery

  return 0; // succcess
}

// Returns the exact positions at joints for the last motion for the exact
// specification of time at that motion
// TimeC = 0.0  ... corresponds to the sitation when cobot was motion is started
// TimeC = 1.0  ... corresponds to the sitation when cobot was motion is finished
// Returns 0 ... on success
// Returns -1 ... on failure, the cobot is moving
// Returns -2 ... on failure, the cobot has not been moved yet since initialization
// It can be sucessfully executed only if the last motion execution is finished.
int
GetExactPositionForLastMotion(char ipaddress[], float timeC, float J[])
{
  if ((timeC<0)||(timeC>1.0))
    return -1; // the time event is out of range

  // TO BE IMPLEMENTED BY CONTRACTOR latest at time of goods delivery
  // - accurate reading of the motion for cobotic arm
  // the hardware accuracy limits that to

  return 0; // succcess
}

// -----------------------------------------------------------------
// END OF API SPECIFICATION


// -----------------------------------------------------------------
// BEGIN OF EXAMPLE USAGE

// -----------------------------------------------------------------
// Auxiliary function
// Generate a random value in range <0,1>
double
R01() {
  return ((double)rand())/(double)RAND_MAX;
  //return drand48();
}

// Set the initial required position of the cobotic arm
void
SetInitialPose(int a, float JB[6])
{
  a=a;
  const float range=270; // minimum range is (-270,+270) degrees
  // This routine can be change for the purpose of testing so
  // physically the cobot does not hit the mounting desk etc.
  // For example, it makes sense to restrict more J[0] and J[1]
  //
  // The principal is simple: generate a new random pose, each
  // time of execution of this program different

  // Set angles at joints
  JB[0] = 0.0 + R01()*range;
  JB[1] = 0.0 + R01()*range;
  JB[2] = 0.0 + R01()*range;
  JB[3] = 0.0 + R01()*range;
  JB[4] = 0.0 + R01()*range;
  JB[5] = 0.0 + R01()*range;
}

// Generate a new pose of the cobot arm
void
GenerateNewRandomPose(const float JB[6], float range, float J[6])
{
  // Set angles at joints
  // This routine can be change for the purpose of testing so
  // physically the cobot does not hit the mounting desk etc.
  // For example, it makes sense to restrict more J[0] and J[1]
  //
  // The principal is simple: generate a new random pose, each
  // time of execution of this program different

  // minimum range by tender specification is (-270,+270) degrees
  const float maxrange=270;

  J[0] = JB[0] - range/2.0 + R01()*range;
  if (J[0] < -maxrange) J[0] = -maxrange;
  if (J[0] > maxrange) J[0] = maxrange;

  J[1] = JB[1] - range/2.0 + R01()*range;
  if (J[1] < -maxrange) J[1] = -maxrange;
  if (J[1] > maxrange) J[1] = maxrange;

  J[2] = JB[2] - range/2.0 + R01()*range;
  if (J[2] < -maxrange) J[1] = -maxrange;
  if (J[2] > maxrange) J[1] = maxrange;

  J[3] = JB[3] - range/2.0 + R01()*range;
  if (J[3] < -maxrange) J[1] = -maxrange;
  if (J[3] > maxrange) J[1] = maxrange;

  J[4] = JB[4] - range/2.0 + R01()*range;
  if (J[4] < -maxrange) J[1] = -maxrange;
  if (J[4] > maxrange) J[1] = maxrange;

  J[5] = JB[5] - range/2.0 + R01()*range;
  if (J[5] < -maxrange) J[1] = -maxrange;
  if (J[5] > maxrange) J[1] = maxrange;
}

// -----------------------------------------------------------------
// Testing functionality of the cobot to be delivered by CONTRACTOR
int
main(int argc, char* argv[])
{
  // Example of the IP address, where the cobot controller is available
  char ipaddress[]="192.168.88.100";

  // Generate enough big array of values to store a single motion
  float *positions = new float[10000000];
  assert(positions);

  // Starts the communication with the cobot controller
  ConnectCobotCommunication(ipaddress);
  if (IsMoving(ipaddress)) {
    cout << "Stop the cobot at the program start" << endl;
    StopCobotMotion(ipaddress);
  }

  // The cobot controller can be in some error state upon startup,
  // such as a failure from the last program execution when a cobotic
  // arm hit an obstacle during motion
  if (IsCobotInErrorState(ipaddress)) {
```

```
  ReinitiateCobotController(ipaddress);
  cout << "Warning: the cobot controller had to be reinitiated,"
     << " some error upon startup" << endl;
}

float Jstart[6], TCPstart[6];
// Get the cobot position at start of the application
GetCobotPosition(ipaddress, Jstart);
ConvertJointsToTCP(ipaddress, Jstart, TCPstart);
cout << "Start cobot arm position" << endl;
for (int i=0; i < 6; i++) {
  cout << "TCP["<<i<<"]= " << TCPstart[i]
     << " J["<<i<<"]= " << Jstart[i] << endl;
}
// kg - example payload 15.0kg
float payloadWeight = 15.0;
// kg.m^2 - example pay load moment of inertia 0.04kg.m^2
float payloadInertia = 0.64;
// m - the distance of payload center of gravity to the center
float payloadDistance = 0.15;
SetWeightMomentOfInertia(ipaddress, payloadWeight,
                   payloadInertia, payloadDistance);

// Set speed of motion
if (1) {
  // Set max speed at joints
  float speedJoints = 0.1; // degrees/s
  SetSpeedJoints(ipaddress, speedJoints);
}
else {
  // Set max speed at TCP and max acceleration and deceleration
  float speedTCP = 10; // m/s
  float accel = 10; // m/s^2
  float decel = 10; // m/s^2
  SetSpeedTCP(ipaddress, speedTCP, accel, decel);
}

// randomize the initial position and random generator
if (1) srand(time(NULL));

// Set the initial base pose of the cobotic arm
// and store it to JB[] array describing the joint angles
float JB[6];
SetInitialPose(0, JB);

// How many TCP poses to be used in this test
int N = 1000;

cout << "This is initial position, now press key"
     << endl << flush;
getchar();

// Now start the loop with N motion steps
for (int i=0; i < N; i++) {
  // Angles at joints
  float range = 25; // range in degrees to generate a new motion pose
  // A new position of the cobot joints
  float J[6];
  if ((i%2)==0) {
```

```
  // Move to the initial position, copy the position
  for (int j=0; j < 6; j++) J[j] = JB[j];
} else {
  // Generate a new pose each second time relative to JB
  GenerateNewRandomPose(JB, range, J);
}

if (1) {
  // Tool Center Point, position + rotation
  float TCP[6];
  // Print out the new position
  ConvertJointsToTCP(ipaddress, Jstart, TCP);
  cout << "i=" << i << " move cobot arm position:" << endl;
  for (int i=0; i < 6; i++) {
    cout << "TCP["<<i<<"]= " << TCP[i] << " J["<<i<<"]= " << Jstart[i] << endl;
  }
}

if (1) {
  // Now compute the planned motion position at required count of time events
  int K2=100; // the number of positions to be checked

  int N=1; // let us plan only one position here
  float timeC[2];
  timeC[0]=1.0; // only one
  float expectedTime;
  int err = SetPositionForTheNextMotion(ipaddress,
            N, // how many positions, at least 1
            timeC, // event times at these positions
            J, // joints setting for the specified times, multiple of 6 values
            expectedTime);
  if (err != 0)  {
  cout << "ERROR: setting the new cobot pose is wrong code= "
       << err << endl;
  continue; // go to the next trial
  }
  cout << "i=" << i << " planning motion OK - expected time for motion is "
     << expectedTime << " seconds" << endl;

  for(int j=0; j <= K2; j++) {
  // normalized time value in range <0.0, 1.0>
  float timeC = (double)j/(double)K2;
  float JT[6], TCP[6];
  // Get planned position at joints for the last motion at
  // normalized time 'timeC'
  GetPlannedPositionForNextMotion(ipaddress, timeC, JT);
  // Analyze and exploit the pose data JT[] - user code by
  // an application ... to be used for checking collision detection!!
  //
  if (1) {
    // Example - convert the exact joint data to TCP and
    // print them to the output
    ConvertJointsToTCP(ipaddress, JT, TCP);
    for (int k=0; k < 6; k++) {
      cout << "j=" << j << "planned J[" << i << "]= "
         << JT[i] << " TCP[" << i << "]= " << TCP[i] << endl;
    }
    cout << "----------------------------------------" << endl;
  }
  } // for j
} // --------------- end of analysis for planned motion ------------------

// Get OS real time in miliseconds, real time, at the start
auto tstart = std::chrono::system_clock::now();
float eps = 2e-5; // specify in meters the convergence condition
// Now start the movement of a cobot arm to a new pose,
// non-blocking operation
int ret=ExecuteCobotMotion(ipaddress, eps);
if (ret) {
  cout << "ERROR - motion was not started err=" << ret << endl;
  continue; // try with another position
}
// Check the position of cobot arm during the motion as frequently as
// possible and store it, there is a lag inaccuracy due to the communication
// via network
int K = 0;
int movingStatus = 0;
// Read as many times as possible during the cobotic arm motion
for(;;) {
  // Read the position for this moment of time
  auto tt = std::chrono::system_clock::now();
  float JC[6];
  std::chrono::duration<double> elapsed_seconds = tt - tstart;
  // returns immediate cobot position at joints at this time
  int movingStatus = GetCobotPosition(ipaddress, JC);
  // store the time and position received from controller
  positions[K++] = elapsed_seconds.count();
  // copy the position to array
  for (int j=0; j<6; j++)
  positions[K++] = JC[j];
  // Is the cobot at the end position of this pose?
  if (movingStatus == 0)
  break; // yes, we can finish the loop
  if (movingStatus < 0) {
    cout << "ERROR: an error occured during the motion from the last pose"
         << endl;
    cout << "The error code is " << movingStatus << endl;
    break;
  }
  assert(movingStatus > 0);
  // Random stop of motion during the execution, with a low probability
  float vrnd = R01();
  const float thresholdStopMotion = 0.01; // probability 0 to 1.0
  if (vrnd < thresholdStopMotion) {
    // Stop the motion immediately, the emergency stop test during motion
    StopCobotMotion(ipaddress);
    break; // break this loop, cobot does not move any longer
  }
} // -------------- end of online recording loop ----------------------

// Get time in miliseconds, real time
auto tstop = std::chrono::system_clock::now();
std::chrono::duration<double> esTotal = tstop - tstart;
cout << "i=" << i << " ... duration of motion took " << esTotal.count()
     << " seconds" << endl;
if (movingStatus < 0) {
  cout << "WARNING: Trying to remove the error state from the motion"
       << endl;
```

```
      ReinitiateCobotController(ipaddress);
    }

    if (1) {
      // Now analyze or/and save the exact data saved during the last
      // motion and use them
       // the number of positions to be checked as recorded by the controller
      int K2 = 100;
      for(int j=0;j<=K2;j++) {
      // normalized time value in range <0.0, 1.0>
      float timeC = (double)j/(double)K2;
      float JT[6], TCP[6];
      // Get exact position at joints for the last motion
      // at normalized time 'timeC'
      GetExactPositionForLastMotion(ipaddress, timeC, JT);
      // Analyze and exploit the pose data JT[] - user code
      // by an application .. to be used by the customer
      if (1) {
        // Example - convert the exact joint data to TCP
        // and print them to the output
        ConvertJointsToTCP(ipaddress, JT, TCP);
        for (int k=0; k < 6; k++) {
          cout << "j=" << j << "recorded J[" << i << "]= "
             << JT[i] << " TCP[" << i << "]= " << TCP[i] << endl;
        }
        cout << "-----------------------------------------------" << endl;
      }
      } // for j
    } // ---------------------- end of analysis for executed motion ------
  } // for i ------------ end of main testing loop ----------------

  // Get the cobot position after it has stopped motion
  float Jstop[6], TCPstop[6];
  GetCobotPosition(ipaddress, Jstop);
  ConvertJointsToTCP(ipaddress, Jstop, TCPstop);
  for (int i=0; i < 6; i++) {
    cout << "TCP["<<i<<"]= " << TCPstop[i]
       << " J["<<i<<"]= " << Jstop[i] << endl;
  }

  // Stop the communication with the cobot controller
  DisconnectCobotCommunication(ipaddress);

  return 0; // end of the main program
}

// ------------------------------------------------------------------
// END OF EXAMPLE USAGE
```